Technical Report ITL-97-2
Supersedes Technical Report ITL-96-4
February 1997

US Army Corps of Engineers
Waterways Experiment Station

A Brief Study of Rational Apex

by Clyde Christopher

Prepared for Headquarters, U.S. Army Corps of Engineers

Technical Report ITL-97-2
Supersedes Technical Report ITL-96-4
February 1997

A Brief Study of Rational Apex

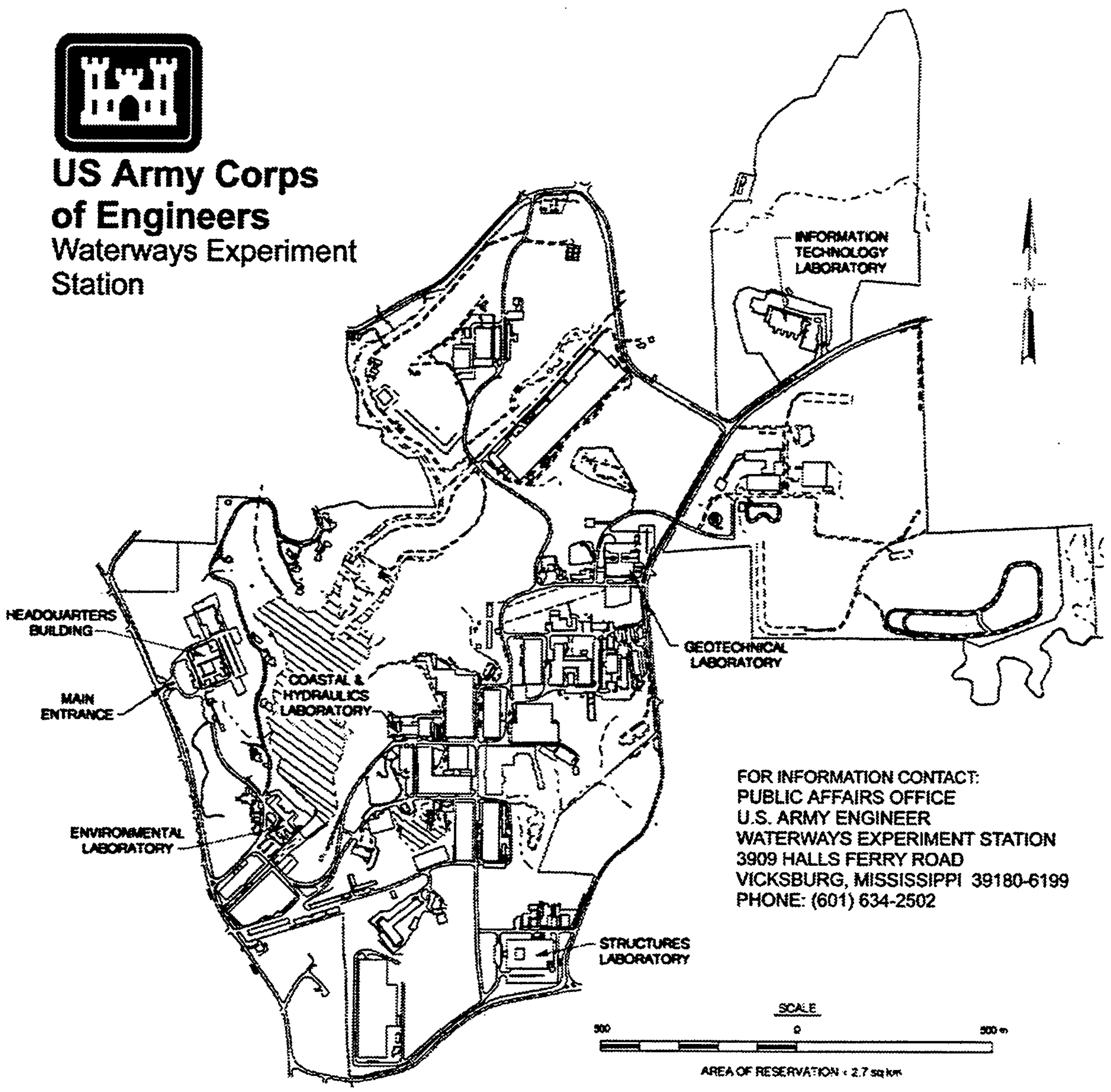
by Clyde Christopher

U.S. Army Corps of Engineers
Waterways Experiment Station
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Final report

Prepared for U.S. Army Corps of Engineers
Washington, DC 20314-1000

US Army Corps of Engineers
Waterways Experiment Station



FOR INFORMATION CONTACT:
PUBLIC AFFAIRS OFFICE
U.S. ARMY ENGINEER
WATERWAYS EXPERIMENT STATION
3909 HALLS FERRY ROAD
VICKSBURG, MISSISSIPPI 39180-6199
PHONE: (601) 634-2502

Waterways Experiment Station Cataloging-in-Publication Data

Christopher, Clyde.
A brief study of rational apex / by Clyde Christopher ; prepared for U.S. Army Corps of Engineers.
36 p. : ill. ; 28 cm. — (Technical report ; ITL-97-2)
Includes bibliographic references.
Supersedes: Technical report ITL-96-4
1. Apex (Computer program)2. Ada (Computer program language) 3. Software configuration management. 4. Software engineering — Management. I. United States. Army. Corps of Engineers. II. U.S. Army Engineer Waterways Experiment Station. III. Information Technology Laboratory (U.S. Army Engineer Waterways Experiment Station) IV. Title. V. Series: Technical report (U.S. Army Engineer Waterways Experiment Station) ; ITL-97-2.
TA7 W34 no.ITL-97-2

Contents

Preface

This report was written by Mr. Clyde Christopher, Professor of Computer Science, Jackson State University, Jackson, MS, under an Inter-Governmental Personnel Agreement. It was originally published as Technical Report ITL-96-4.

The author expresses sincere appreciation to Dr. N. Radhakrishnan, Director of the Information Technology Laboratory (ITL), U.S. Army Engineer Waterways Experiment Station (WES); Dr. Windell F. Ingram, Chief, Computer Science Division, ITL; Mrs. Deborah Dent, Group Leader, Research Group, Computer Science Division; and Mr. Alan Lee, DynTel Corporation, for the opportunity to engage in this research project. The continuous development and use of advanced processes at ITL are the result of their insight and their quest for excellence in the field.

A hearty thanks is extended to Mr. Richard Carpenter, Contractor, DynTel Corporation, for equipment and systems technical support throughout this short period. Without his assistance nothing could have been accomplished.

During the publication of this report, Director of WES was Dr. Robert W. Whalin. Commander was COL Bruce K. Howard, EN.

1 Introduction

Two important features of **Apex** are **subsystems** and **views**. A subsystem is a section of a larger system. A rational subsystem is a directory with a **.ss** filename extension.

A view is a subdirectory inside a subsystem directory. It provides a means of grouping certain files, or objects, that constitute a project. There are two types of views in **Apex**. **Working** views have a **.wrk** filename extension while **Release** views have a **.rel** filename extension. **Working** views contain objects being currently developed and **Release** views store release versions of software systems.

Apex uses the Rational Configuration Management and Version Control (CMVC) system to help you manage development and simultaneous changes in objects within a project. CMVC supports a check-in/check-out reservation system for the source files in a project and maintains a change history for each object in the project.

The **Apex** Ada editor helps you to generate code in Ada more efficiently. It includes editing, pretty-printing, syntactic analysis, semantic analysis, and traversal browsing. It forces your programming team to follow a consistent programming style, increasing productivity, and reducing maintenance problems.

In managing compilation **Apex** uses switches to set view specific options for the **Apex** compiler and other tools, automatically determines the dependencies between Ada units so a makefile is not needed to control compilations, and uses its awareness of dependencies between statements and declarations within Ada units to perform optimal recompilatiom automatically when a change, deletion, or addition is made. Syntactic and semantic analysis highlights errors and supplies error messages that help you correct your program immediately.

The Ada **debugger** makes it easy for you to analyze the behavior of an Ada program as it runs. It allows you to display the source code for any part of a program, display the contents of the stack for any task in a program, display and modify the values of variables in a program, place breakpoints in the program, display tasks and their current execution state, execute one statement,call, or machine instruction at a time, and handle exceptions in various ways.

Getting Started

When **Apex** starts, the following windows will appear:

a. An **Apex** panel window

b. Directory-viewer window

c. Version.doc window

d. Debugger Log window

The **Apex** panel window allows you to display the **Project Setup** window, used to create projects, visit files using the directory-viewer window, display the **Windows** window, or exit **Apex** terminating your **Apex** session.

The **directory-viewer** window displays the name of a directory and its contents. It is the main window used to access **Apex** files and directories.

The Version.doc window provides copyright, trademark, and technical support information and contains buttons to access that information. The **Debugger Log** window is iconified when **Apex** starts. It will be discussed later.

Creating Objects

You can create, copy, open, save, and delete objects by selecting commands from the **File** menu. To create an object click **File:New** from the directory viewer **File** menu. **Apex** will open a cascading menu offering the following choices:

a. New Ada

b. New C++

c. New Directory

d. New History

e. New Export Set

f. New View

g. New Subsystem

h. New Configuration

Clicking on either of these will cause **Apex** to open a dialog box designed for your choice. Inserting the proper information and clicking **OK** or **Apply** will create your object.

2 Text Editors

Any UNIX editor may be used to edit text in the **Apex** environment. However, **Apex** provides two editors, the **Apex** text editor and the **Apex Ada** editor, which support mouse-driven editing operations. Of course, the **Apex Ada** editor has special features for editing **Ada** source files.

The Apex Text Editor

The **Apex** text editor is used to create, view, and edit source code and other types of text files. It is started by selecting the **File:New:New File** command from the **File** menu. An existing file may be opened by choosing the **Open** command from the **File** menu. Either action causes an **Apex** text editor window to appear on your screen. The window will have a menu bar which contains **File, Edit, View, Navigate, Control, Compile, Debug, Tools,** and **Help** buttons. Clicking on **Edit** brings up a menu with commands to perform standard delete, cut, copy, and paste operations; undo editing; reinstate editing commands that have been undone (redo); search text for specified patterns; and other editing operations. Selecting these with the mouse is much faster than using the keyboard.

The Ada Editor

The **Ada** editor is used to create, view, and edit Ada source code. An **Ada** editor window appears as result of each of the following:

a. Selecting the **File:New:New Ada** command from a file menu.

b. Selecting **Open** from a file menu (**Open** dialog box will appear).

c. Selecting the unit in an **Apex** directory-viewer window and choosing **Visit**.

The **Ada** editor window contains a menu bar with buttons that generate menus for a variety of operations. The **Edit** menu allows you to perform all of those operations available in the **Apex** text editor. The **Compile** and **Navigate** menus offer you commands for many Ada-specific editing features.

The **Ada** editor recognizes the structure of Ada elements and their relationships with each other. So it offers special browsing, selecting, formatting, and error-checking commands for Ada units. You can browse through independent Ada units when you have need to:

a. Inspect someone else's program and want to find the type definitions for program variables.

b. Debug a problem and want to see where and how a particular subprogram is implemented.

c. Consider a change to a subprogram and want to know exactly where the subprogram is used.

d. To traverse Ada elements simply place the cursor on an element and select **Visit** or **Visit In Place** from the **Navigate** menu. The results are as follows:

e. **Navigate:Visit** or **Navigate:Visit In Place** - Traverses to the defining occurrence of a program element.

f. **Navigate:Visit Ada Name** - Opens a dialog box you can use to traverse to an Ada unit, given its name.

g. **Navigate:Visit Body** or **Navigate:Visit Body In Place** - Traverses to the body of an Ada specification.

h. **Navigate:Other Part** or **Navigate:Visit Other Part In Place** - Traverses to any selected part of an Ada specification.

The **Ada** editor lets you find all places where a particular Ada element is used by selecting one of the commands

Compile:Show Usage

Compile:Local Usage

It also lets you find all Ada elements that are not used by selecting

Compile:Show Unused

In each case the elements in question are highlighted. You can traverse from one occurrence to another by clicking the **Message ->** button or the **Message <-** button.

Several commands available from the **Compile** menu make formatting of Ada units easier and more convenient.

a. **Compile:Build Body** creates a skeletal Ada body for an existing Ada specification.

b. **Compile:Syntax** does syntactic completion and pretty-printing. Syntax completion includes insertion of prompts for missing units, insertion of ending punctuations, insertion of reserved words, and matching identifiers in the **end** statements of loops. Pretty-printing includes capitalization of identifiers, adjusting line breaks, adjusting indentation, and spacing around delimiters and operators.

c. **Compile:Pretty-Print** prints an Ada unit that was modified with some other editor.

d. **Compile:Complete** does semantic completion such as supplying the **with** clauses, **use** clauses, renames declarations, and parameters for procedure calls.

e. **Compile:Make Inline** absorbs a subunit into the subunit's parent.

f. **Compile:Make Separate** converts a selected subprogram, task, or package body into a separate unit.

Error checking is made easy by the **Ada** editor. The **Compile:Syntax** menu command performs syntactic checking. The **Compile:Analyze** command performs semantic and syntactic checking. **Apex** highlights any errors found. You can traverse from one error to another by clicking the **Message ->** button or **Message <-** button. Upon clicking the **Explain** button **Apex** displays a window with the error message.

3 Writing Programs in Ada

An Ada application typically consists of many interdependent units such as procedures, packages, and functions. **Apex** is designed to support the development of large complex systems that are divided into subsystems to make them more manageable. All compilations in **Apex** are done in the context of subsystems regardless of size of the program.

When **Apex** starts it opens a directory-viewer window that displays a list of files and directories. Directories appear before files. Names beginning with numbers precede names beginning with letters of the alphabet. Names beginning with uppercase letters precede names beginning with lowercase letters. To view an object listed in the directory-viewer window you can select the object and click **View** or you can double-click the object's name. You can also issue commands from the directory-viewer window's menu to load, save, and print files and to execute and debug programs.

Creating a Subsystem

A subsystem may be created for personal use or for a group project. If you create a subsystem for your personal use, you can place it in your home directory where it is convenient for you to use. If you create a subsystem for a group project, you should place it in a common area accessible to each member of the group. To create a subsystem:

a. From a directory-viewer window, select the **File:New:New Subsystem** menu item.

b. **Apex** then opens the **New Subsystems** dialog box.

c. Enter the full pathname for the new subsystem in the text field labeled **Subsystem Full Name.** (It is not necessary to enter the .ss extension.)

d. Click on **OK** or **Apply**.

Once your subsystem is created you must have views within the subsystem. Views contain files with programs. There are two kinds of views: working views, in which programs are developed, and release views which store release views of the subsystem. To create a view:

a. From the **File** menu in a directory-viewer window select **File:New:New View.**

b. Apex will then open the **New View** dialog box.

c. Make sure the name of the appropriate subsystem is in the **Subsystem** field. Enter a name for your new view in the **Name** text field. (**Apex** will automatically add a **.wrk** suffix for a working view or a **.rel** suffix for a release view.)

d. Select **working** or **release** in the block opposite **Create an Empty View.**

e. Click the **Visit It** check box.

f. Click on **OK** or **Apply.**

Ada source files are produced in a working view. The specification of a source file is always the name of the unit with the suffix **.1.ada** added. The body of a source file is always the name of the unit with the suffix **.2.ada** added. An Ada source unit is created as follows:

a. From a directory-viewer window, select the **File:New:New Ada** menu item.

b. **Apex** will open a **New Ada** dialog box.

c. Click the **Package** box and select from the pop-up choices of **Package, Procedure, Function, Task,** or **Protected.**

d. Click the **Spec** box and select from the pop-up choices of **Spec, Generic Spec, Body,** or **Subunit.**

e. Enter a name for your unit in the **Name** text field. (**Apex** will automatically add the **.ada** suffix.)

f. Check the **Visit It** box.

g. If you want your procedure to be controlled under the Rational CMVC system, check the **Make It Controlled** box.

h. Click on **OK** or **Apply. Apex** opens an Ada editor window for a new source file. The principal delimiters for your Ada unit will appear in the window.

Compiling a Program

A program is always compiled within the compilation context of its enclosing views. The compilation context of a view includes the compiler, the values of the compiler's switches, and any imported views that provide external visibility to the units being compiled.

Apex provides commands for compiling and linking Ada programs from the **Compile** menu in the Ada editor window. The compilation commands and their effects are given below.

a. **Syntax** checks the current unit for syntax errors and then pretty-prints the image. It will attempt to correct any errors found. Objects and delimiters representing errors are displayed in red print. Error messages are displayed in the **Message** window. You may traverse to these by selecting **Message ->** or **Message <-.**

b. **Pretty Print** adjusts capitalization of identifiers, line breaks, indentation levels, and spacing around Ada-specific delimiters and operators. It reformats code prepared on other editors.

c. **Semanticize** performs a semantic analysis of the current unit. It updates the unit's underlying structure with semantic information and verifies that the underlying structure conforms to the semantics rules of the Ada language.

d. **Parse** parses the contents of the current program unit.

e. **Analyze** performs semantic analysis on the contents of the current unit. It checks whether the unit is a legal Ada compilation unit.

f. **Code** generates object code from the current unit and advances the unit to the coded state.

g. **Link** generates an executable from the current unit. That includes all specs and bodies that are accessible through **with** and **subunit** relationships.

h. **Import Text Files** opens a dialog box that lets you import a text file and convert it into an Ada source file.

i. **Clean** removes compilation artifacts to what is required for the specified state.

j. **Show Errors** displays a dialog box that lets you search for Ada units with errors.

k. **Local Usage** displays the occurrences of a selected declaration within the current library unit.

l. **Show Usage** displays a viewer window that lists the instances of a selected object with a specified scope.

m. **Show Unused** opens a window that displays the name of units containing declarations that are unused in the specified closure.

n. **Show Demotion Impact** opens a dialog box that displays what other units will be affected if you change the compilation state of the current unit.

o. **Build Body** incrementally builds bodies needed by selected units.

p. **Make Separate** makes a selected entity within a unit a separate subunit.

q. **Make Inline** makes a subunit part of its parent unit.

Linking a Program

The **Compile:Link** command links the current selection, producing executables based on main entry points in the designated program units. Any parameterless procedure can serve as the root of an Ada program that can be linked to produce an executable. If linking is successful, the units advance to the linked compilation state.

When you choose **Compile:Link** from a viewer window **Apex** displays the **Link** dialog box which lists the selected unit and allows you to list additional units. If you choose **Compile:Link** from an editor window, **Apex** analyzes the current unit using the **Link** dialog box defaults.

Executing a Program

When you choose the **File:Run** menu item, Apex will open the **Run** dialog box. The **Run** dialog box contains three text fields where you may enter the name of the program you want to execute, arguments that you want to pass to the program, and the context in which the program will execute. The execute field and context field are usually filled by **Apex**.

The **Run** dialog box also allows you to redirect input, output, and error output. If you select the **Debug** option, **Apex** will automatically select the **Direct output to an xterm** option. Clicking **OK** in this box causes execution of the program.

4 Debugging

When you start **Apex**, a debugging window called the Ada Debugger Log window automatically opens in the form of an icon on your screen. When you conduct a debugging session, all interactions with the debugger are displayed in a sequential log in the Debugger Log window. You may raise the window by double-clicking on it or you may select the **Debug:Window:Main Log** menu item in an Ada editor window.

Debugging is an extremely important step in software development. Researchers have pointed out that a high percentage of computer programs have errors when first run. However, a lower percentage have logical errors. Debugging may be a part of the entire compile and test procedure.

Starting the Debugger

The easiest way to start a debugging session is by selecting the **File:Run** menu item from either a directory-viewer window, the **Debugger Log** window, or an Ada editor window. The **Run** dialog box opens. You can execute and debug the program simultaneously by selecting the **Debug** button and clicking **OK** or **Apply**.

When you start debugging a program, **Apex** creates a new job. You have two jobs visible within **Apex**: the debugger server that monitors the debugger functions and a job that executes the program. When you debug an Ada program, source-level debugging is available for all readable code in the closure of the program. If the debugger halts execution of the program, you can select the **Visit-Source** option and **Apex** will display the current location in the source code in an editor window.

Once the debugger is started. program execution is under your control. You can let the program run normally or you may stop execution at predetermined points or when predetermined conditions occur. Some of the things that you can do are given below.

You can run the program one step at a time by choosing **Step Statement** from the **Step** submenu. Execution will halt after each declaration is elaborated and after each statement is executed. Stepping through a program lets you analyze its behavior.

You can step through the program one statement at a time, stopping inside procedure calls but not stopping in function calls. This can be done by choosing the **Step Into** option from the **Step** submenu.

You can step through the program within the current subprogram one step at a time without stopping inside any subprogram calls. To step through a program this way choose the **Step Over** option from the **Step** submenu.

You may step through the program one instruction at a time by choosing the **Step Instruction** option from the **Step** submenu.

You can run the program until the current stack frame is exhausted, and then stop before executing the next Ada statement. This is done by choosing **Run Returned** from the **Step** submenu.

Finally, you may choose to continue program execution after examining the reason for a pause by selecting **Continue** from the **Debug** menu. After stepping operations are completed execution may be returned to normal by selecting **Debug:Clear Stepping**.

To the debugger, every program consists of one or more tasks. The main program is called the root task. Apex assigns to each task a number. Whenever the debugger displays information about a task it displays that number. The task number may be used to issue commands for debugging.

When a program consists of multiple tasks, each task maintains its own call stack. If a function or procedure is called, local variable definitions and other important information are placed in a construct called a stack frame. When the **Apex** debugger stops a task, this information can be retrieved from the stack by the debugger.

You can display the status of all tasks in a program by selecting **Debug:Window:Tasks** from the debug menu. You can display the contents of stacks by selecting **Debug:Window:Stack**. A task is always in one of two states: running or stopped. When a task is in the stopped state it does not execute. All other tasks are listed as **Running**, even though they are actually stopped. This is done so you can tell which is the current task.

During a debugging session there are two jobs being executed: the debugger job and the program job. When your program is loaded, **Apex** automatically creates a program job and a debugger. The debugger controls execution of your program. Hence, the debugger job controls the program job. The Jobs window shows the status of the program job, while the **Ada_Debugger_Server** or **Servers** window displays information about the debugger job.

Setting Debugger Options

When you select the **View:Options** command from any debugger window, the **Ada Debugger Options** dialog box is displayed. You can customize the debugger to fit your needs by selecting options. These options and their default settings are listed here:

a. **Auto_Kill** automatically kills the current job being debugged before running a new job. Default is False.

b. **Display_Addresses** shows the program counter location at the stopping point and in the Stack window. Default is True.

c. **Visit_Source** automatically displays the source when execution stops. Default is True.

d. **Save_Exceptions** saves the current exception-handling requests in the debugger state file when the program is terminated. Default is True.

e. **Save_Breakpoints** saves the current breakpoints in the debugger state when the program terminates. Default is True.

f. **Save_Options** saves your options in the debugger state file when the program terminates. Default is False.

g. **Read_User_State** reads the debugger user-state file when a program is loaded. Default is True.

h. **Read_Program_State** reads the debugger program-state file when the program is loaded. Default is True.

i. **Qualify_Names** displays full debugger pathnames for all names except those in the Stack window. Default is True.

j. **Qualify_Stack_Names** displays full debugger pathnames in the Stack window. Otherwise, relative debugger pathnames are displayed. Default is True.

k. **Special_Type_Display** specifies that you want to use the customized display associated with this object, if one exists. Default is True.

l. **Display_Levels** sets the default number of levels to display in the **Show Data** dialog box. Default value is 4.

m. **Expand_Pointers** sets the default to selected for the **Expand Pointers** option in the **Show Data** dialog box so pointer objects are dereferenced. Default is True.

n. **Show_Location** sets the default to selected for the **Show Locations** option in the **Show Data** dialog box so that the memory location or register number is also displayed, if applicable. Default is False.

o. **Element_Count** sets the number of elements to display for an array. Default value is 10.

p. **Stack_Count** sets the number of call levels to display in the Stack window. Default value is 10.

When the **Ada Debugger Options** dialog box is displayed, the default settings will be present. You simply need to click on those that need changing.

Using Breakpoints

One way to monitor the behavior of a program is to set breakpoints in the program. A breakpoint may be set at any declaration or statement in the program. A breakpoint is in one of two states, active or inactive. If it is inactive, it has no effect on the execution of the program. If it is active, it will halt execution of the program when it reaches the point where the breakpoint is placed.

There are two types of breakpoints. A permanent breakpoint remains active for as long as a program is running under the debugger or until it is explicitly removed. A temporary breakpoint is automatically removed after the first time it is encountered.

There are three ways to create a breakpoint. The easiest way is to open an editor window and select the desired breakpoint location. Then choose the **Debug:Break Here** command. You may also open the **Breakpoints** window and choose **Breakpoints:New** or open any debugger window and choose **Debug:Break**. Either of these commands will cause the debugger to display the **Set Breakpoint** dialog box. The breakpoint can be set by entering the statement or declaration in the location field and selecting options as needed.

You can open the **Breakpoints** window by opening the **Debugger Log** window and selecting **Breakpoints** in the windows menu or by clicking the **Breakpoints** button in the **Debugger Log** window's button bar. All active breakpoints are displayed in the upper half of the window and inactive breakpoints are in the lower half.

To activate an inactive breakpoint:

a. Open the **Breakpoints** window.

b. Select the breakpoint you want to activate.

c. Choose the **Breakpoints:Activate** command or click the **Make Active** button in the **Breakpoints** window.

To deactivate an active breakpoint:

a. Open the **Breakpoints** window.

b. Select the breakpoint you want to deactivate.

c. Choose the **Breakpoint:Deactivate** command or click on the **Make Inactive** button in the **Breakpoints** window.

To remove a breakpoint:

a. Open the **Breakpoints** window.

b. Choose the breakpoint you want to remove.

c. Choose the **Breakpoint:Remove** command.

When the debugger has halted execution of a program, you can step through the statements, instructions, expressions, functions, and procedures in various ways. Selecting the **Debug:Visit_Source** option opens an Ada editor window that displays the current Ada unit. The current statement is selected. The stepping commands are as follows:

a. **Debug:Step:Step Over** from an Ada editor window or **Execution:Step Over** from a debugger window - Step to the next statement in the same context.

b. **Debug:Step:Step Into** from an Ada editor window or **Execution:Step Into** from the debugger window - If the current statement is not a procedure call, step to the next statement in the same context. If the current statement is a procedure call, step to the first statement of the procedure.

c. **Debug:Step:Step Statement** from the Ada editor or **Execution:Step Statement** from a debugger window - Step to the next source statement regardless of its containing unit. Each function call and each procedure call is stepped through statement by statement.

d. **Debug:Step:Step Instruction** from the Ada editor or **Execution:Step Instruction** from the debugger window - Step one machine instruction.

e. **Debug:Step:Repeat Step** from the Ada editor or **Execution:Repeat Step** from a debugger window - Repeat previous step command.

f. **Debug:Continue** from the Ada editor or **Execution:Continue** from a debugger window - Continue execution after the program has stopped for any reason.

g. **Debug:Step:Run Returned** from an Ada editor or **Execution:Run Returned** from a debugger window - Execute until the current stack frame is completed and stop execution before the next Ada statement.

h. **Debug:Stop** from an Ada editor or ***Execution*:Stop** from a debugger window - Halt the program being debugged.

i. **Execution:Clear Stepping** from a debugger window - Clear stepping from all threads.

Displaying Values of Objects

Apex provides two **Debug** commands you can use to display the current value of a selected object or expression. The **Debug:Display** command from an Ada editor window displays a specified object in an elided form in an **Object Display** window.

The **Debug:Show** command operates in different ways depending on whether you issue the command from an Ada editor window or from a debugger window. When issued from an Ada editor window, it displays the current value of any object or expression that is selected.

When issued from a debugger window, **Debug:Show** opens the **Show Data** dialog box. In the **Expression** text field you list an expression whose value you want displayed. For an object you must enter the debugger pathname.

During a debugging session you may monitor objects by displaying their values in the **Monitors** window. The **Monitors** window is displayed by selecting **Windows:Monitor** from a debugger window or **Debug:Window:Monitors** from an Ada editor window.

Machine Level Debugging

An essential aspect of debugging is having the ability to determine the location when the program stops and the value of data being operated upon by the current instruction. Also, it is important to know the contents of registers at that moment. **Apex** lets you perform machine-level debugging by displaying and modifying registers and memory locations.

There are two ways to obtain information. The **Registers** window displays the current values of all registers in the current focus. When the **Floating Point Registers** option is selected, the floating point registers are also shown. The **Registers** window can be displayed by executing the **Windows:Registers** command from a debugger window or **Debug:Window:Registers** from an Ada editor window.

In the **Debugger Log** window you can display the memory location or register name containing a specified object by the following:

a. Execute the **Debug:Show** command.

b. When the **Show Data** dialog box appears, set the **Show Location** option.

c. Click **OK** to close the **Show Data** dialog box.

d. Execute the **Registers** command. When **Apex** displays the **Registers** window, the memory location or register name containing the selected object is displayed in the **Debugger Log** window.

When you execute **Debug:Modify Register** from an Ada editor window, **Apex** will open the **Modify Register** dialog box. Enter the register name in the **Register** field. The **Task** field may be filled in. However, the default is the present focus. The new value must be entered into the **New Value** field. This may be a string or a based number. Hex is the default. After you click **OK** or **Apply,** an appropriate message will appear in the **Debugger Log** window.

Contents of memory locations may be displayed in two ways. If you select the **Show Locations** option in the **Show Data** dialog box and request information about an object that is not stored in a register, the location and value of the object will be displayed in the **Debugger Log** window.

You may also execute **Windows:Memory** from a debugger window or **Debug:Window:Memory** from an Ada editor window to display the **Memory** dialog box. Since the **Memory** dialog box is empty, **Apex** will also open the **Memory Bounds** dialog box. You can enter the beginning location and range of memory locations you wish to display. The range is entered in multiples of 16 bytes. The location and contents will be displayed in the **Memory** window.

To modify the data in a memory location, execute the **Debug:Modify Memory** command. **Apex** will display the **Modify Memory** dialog box. The **address** field must be filled in hexadecimal. In the **Count** field enter the number of bytes to be changed. the **New Value** field must be filled with a string or a hex number or a based number. Hex is the default. The new value will be displayed in the **Memory** window.

Debugger Windows

When the debugger is created, it automatically creates a **Debugger Log** window in which to record the status of the debugger and the program being debugged. All interactions with the debugger are displayed in the **Debugger Log** window as a sequential log. The debugger echoes all commands in this window as they are executed and commands that generate output display it in this window. At the beginning of execution several messages are displayed. The first message identifies the program that you are running under the debugger. The next set of messages show which breakpoints are installed. The final message indicates that the debugger is stopped at the creation break. As you debug the program, a message is added to the **Debugger Log** window to record each debugger operation. Text in the **Debugger Log** window can be modified using the **File:Edit** command and it can be selectively copied into other windows using the **Edit:Copy** command. You can split the window into two display areas using the **View:Split** command, or clear the **Debugger Log** window using the **File:Clear** command.

The **breakpoints** window displays the list of currently defined breakpoints. The window has an active area that displays the active breakpoints and an inactive area that displays the inactive breakpoints. Each breakpoint has the following format:

breakpoint number: location in task(s)

The **Stack** window displays the frames in the execution stack with the current focus being displayed at the top. Each frame row has the following format:

frame number PC = # PC address: statement location

The **Exceptions** window displays the current catch and propagate exception-handling requests. Each exception-handling request has the following format:

Catch	*Exception name*		*Location*		*Task(s)*
or	*or*	*at*	*or*	*in*	*or*
Propagate	*All*		*All*		*All*

The **Registers** window displays the registers and their current values. If the **Floating Point Registers** button is checked, the floating point registers are also displayed. Each row of the display gives the register name, the hex register value, and the decimal register value.

The **Memory** window displays the contents of specific memory locations. Each row contains a memory address of the first byte and the contents of four consecutive 4-byte groups.

The **Assembly** window displays the assembly language instructions for the program currently executing. The format of an assembly instruction calls for an address, machine level instruction, assembly instruction, and source code.

The **Task** window displays the tasks which compose the currently executing program. Each line contains a thread, a thread designator, and a state.

The **Monitors** window displays the names of all objects and expressions being monitored. When execution stops, the debugger displays the current value of all currently active monitored expressions in the **Debugger Log** window.

The **Units** window displays the library-level Ada units comprising the currently running program in alphabetical order. A filename and a full view name is given for each.

The **Input/Output** window accepts input and displays program output.

The **Object Display** window displays the value of a selected expression. It allows the interactive expansion and elision of the subcomponents of a structured object.

5 Managing Large Projects

Apex subsystems are modules. Subsystems support abstraction, encapsulation, modularization, and the reuse of code. There are a number of well understood principles for decomposing systems into subsystems. However, there are many issues to consider, including the following:

a. The nature of the project being undertaken.

b. The organization that is producing it.

c. The size of the project.

d. The number and kind of deliverables.

e. The technology being used in the project.

f. The geographical distribution of the organization producing the project.

Decomposition of the System

The main reason for decomposing a system into subsystems is to control and manage complexity. Each of the smaller pieces of the system can be more easily understood than the entire system because it is smaller. The system can be understood in terms of its major pieces and their interfaces while ignoring the internal details of each subsystem. The person or team who makes the decomposition must understand the entire system, including the details of each subsystem.

Another reason for decomposition is to make it possible for many developers to work on a system in parallel. In order to do this, the work must be divided so that developers can work without constantly having to edit the files of others. There must be common agreement on interfaces. Once this is decided, each developer can implement and test his part of the system independently of others.

Other reasons for decomposing a system are:

a. To facilitate reuse of code. A subsystem may be reused for, or obtained from, another similar system.

b. To facilitate testing. Each subsystem can be tested independently of the other parts.

c. To isolate dependencies on the execution platform, devices, or other factors likely to change.

Subsystem decomposition also supports variants of a system. A variant is a version of a system that has some different characteristics but is mostly the same. **Apex** easily handles this under version control.

There are very general patterns that commonly occur in systems. More than one organization might be applicable to a given system. A system can be decomposed into subsystems using layering. Abstractions in the system are placed into a hierarchy based on their use of other abstractions. In the bottom layer are abstractions that are self-contained. In the next higher layer are abstractions that make use of only the layer below them, and so on. Thus, there is a hierarchy formed based on using relationships between subsystems. **Apex** can represent and enforce a hierarchical layered architecture.

A complex system may perform several different functions for different clients. There can be multiple applications in a system from a client's point of view. A common architecture for building such systems is to have a layered infrastructure that is common to several vertical applications. The vertical applications themselves are layered internally. The vertical applications may import one or more subsystems in the infrastructure layer, but do not import subsystems in other vertical applications.

Subsystems and Views

A view contains the files that represent the elements in a subsystem. Each view represents an alternate implementation or an alternate release of its enclosing subsystem.

If files in different views have the same view-relative name they are associated with the same logical element in the subsystem and they are said to be corresponding files.

Each view provides both a version control context and a compilation context to support the development and compilation activities involving the files in the view. In its role as a version control context, a view provides:

a. A way to specify the desired version for each element in the subsystem.

b. A way to generate new versions and new objects.

c. Methods to interchange versions with developers working in other views, and ways to limit the interchange when the work of different developers should not be mixed.

To provide a compilation context, a view manages:

a. The compiler that will be used to compile program units in the view.

b. Switches that are used by the compiler.

c. The visibility to program units in other subsystems.

d. The visibility that the view itself will provide to clients in other subsystems.

e. Manipulation of all compiler generated files.

Each view of a subsystem is populated with files that represent the contents of the subsystem. Each file in a view represents a specific version of some element of the subsystem. More than one view can exist in a subsystem. Each view can be used for a variety of purposes. The purpose is determined mostly by convention, but **Apex** makes a distinction between a working view in which development is done, and a release view that is used to represent a frozen release of a subsystem.

Often each developer will have his own view of each subsystem on which he is working. Developers check out and check in files, edit, compile, and debug using their own views.

Views can be created that are used for integration or construction of releases of a system. When releasing a system, a consistent set of views of all the subsystems needs to be built.

Views can be created to serve as a repository for a specific version of a system or of some subsystems. This might be for a stable version for some clients to use or for testing purposes.

Views can also, be created to represent variants of a system The contents of files in such a view may be different from the corresponding files in a view for a different variant of the system. Variants might be created to support different hardware platforms. **Apex** allows you to specify the history name for each file in each view. Files are considered to correspond only if they have the same view-relative name and the same history name.

Creating Subsystems, Views, and Towers

To begin a large project a directory must be established for the project with a permanent name and accessible to all workstations to be used in the project.

Next, create all subsystems, within the directory, to be used in the project as follows:

a. Select **File:New:New Subsystem** to bring up the **New Subsystem** dialog box.

b. Fill in the full name of a subsystem to be created.

c. Since you will be creating a number of subsystems, click on **Apply** to keep the dialog box open.

d. Enter the name of the next subsystem.

e. Click on **Apply,** and continue until all subsystems have been created.

A set of views will need to be created, at least one per subsystem. The executables for the system will be built within these views. A naming convention should be used for choosing consistent sets of views. You should use the same view name for a view in each subsystem as an indication that these views are intended to work together for a common purpose.

Such a consistent set of subsystems is called a tower. For instance, a set of views in which the first version of the project is created, edited, compiled, and debugged will be a tower. There will normally be a need for several towers for different purposes. A tower will be needed for testing the system. **Apex** has the capability to create and manage multiple towers. It is not necessary to keep each tower you want. Towers can be created and populated from the CMVC database on demand.

There may also be a need for personal towers for each developer or for each development group. This gives them complete control over changes in that tower.

To create a view, select **File:New:New View.** This brings up the **New View** dialog box. The simple view name and subsystem name are to be entered. If the default model is not to be used, then the model field must be filled. Since you will be creating several views, turn off the **Visit the new view** toggle. Now, click on **Apply** to create the view.

In order to reduce the labor of creating all views in several towers, a single tower can be created with imports set up in the tower. Then the tower can be copied to create all other towers.

Imports

Imports are the property of a view. Select **Control:View Properties** to bring up the **View Properties** dialog box. To set the imports for each view:

a. Fill in the view name.

b. Press **Return** or click **Reset** to show the current view properties for that view.

c. Type the full name of each view to be imported in the **Add** field of the Imports area and press **Return** or click **Add.** Repeat until the imports list field shows the complete list of imports desired for the view.

d. Click **Apply** to set the imports for the view.

If there are a number of subsystems that need to have the same views imported, this can be done as follows:

a. Select **Control:Change View Properties** to display the **Change Properties Of View** dialog box.

b. Add the views whose imports are to be changed to the **Change Properties of Views** list field at the top of the dialog box.

c. Press **Return** or click **Add.**

d. Click the **Change Imports** button.

e. Click **Add/Replace** to specify the imports are to be added.

f. Add the names of the views to be imported to the **Change Properties** list field.

g. Press **Return** or click **Add.**

h. Click **OK** or **Apply** to add the listed imports to each of the listed views.

There may be a need to have certain views within a system import each other. This mutual import relationship may be established by starting with the **Change View Properties** dialog box as follows:

a. Select **Control:Change View Properties** to display the **Change View Properties** dialog box.

b. Add the names of the views to be imported into each other to the **Views** list field by entering their names in its **Add** text field and pressing **Add.**

c. Check the **Change Imports** box.

d. Click the **Import Each Other** button in the **Change Imports** area.

As we stated earlier, once a tower has been set up, you can create additional towers by copying it. The **File:Copy Object** command brings up the **Copy Object** dialog box. The process can be done as follows:

a. Click the **Copy Views** button.

b. Add the list of views to be copied to the **Copy Views** list.

c. Click **Add** or press **Return.**

d. In the **Destination** area, click **Destination name is subsystem-relative view name.**

e. In the **Destination** area, enter the new tower name in the **Name** field.

f. Click **OK** or **Apply** to copy the tower.

Working in a Tower

The **File:New** menu provides a cascade which has operations for creating a number of kinds of objects. It is used for creating program units and text files. Usually new objects will be placed under configuration management control. This can be done automatically using **Apex.**

Apex gives you the ability to control changes to individual objects in a subsystem, and track what changes were made, when, why, and by whom. Each object under CMVC control is associated with a history of changes. You can construct, release, and maintain multiple consistent sets of versions in each subsystem. Each alternative set constitutes a view of the subsystem. At a higher level you specify a configuration of views from each subsystem to create a complete system.

When a file is placed under version control, **Apex** starts tracking the file's development in a special CMVC database. In this database, **Apex** saves each version of the file as it is developed. In this way, the CMVC database provides each developer with access to the same set of files, even though each file may be under a different stage of development.

When an object is controlled, you must check it out before you can modify it. When you have checked it out, you can edit it in any way you like, but all other users are locked out. They cannot modify the object until you have checked it in again.

Every time a file is checked out, modified, and checked in, a new version is created in the CMVC database. Each version represents a snapshot of the file at some moment in time. When files are checked out, modified, and checked in repeatedly, the result is a time-ordered, numbered sequence of versions, called a *version history*. The CMVC database maintains version histories by recording all changes made to every controlled object. You can use the CMVC database to see any version in a file's version history.

To create a new Ada unit:

a. Click on **File:New:New Ada** to display the **New Ada** dialog box.

b. Set the option menus.

c. Enter the name for the new unit.

d. Check the **Place Under CMVC Control** checkbox.

e. Click on **Visit It.**

f. Click on **OK** to display the Ada editor window.

Apex will append the proper suffix, **.1.ada** for an Ada spec or **.2.ada** for an Ada body.

Again, an object in **Apex** under CMVC control must be checked out in order to edit it. An object may be checked out in several ways:

a. Select **File:Edit** in an **Apex** editor window (or click on the **Check Out** toggle button in the button bar). The **Check Out** dialog box for the unit is displayed. Click **OK.**

b. Select the object in a directory viewer window and click on **Control:Check Out.** This brings up the **Check Out** dialog box. Click **OK.**

c. Select the object in a directory viewer window and click on **Control:Object Properties.** This will display an **Object Properties** dialog box for the object. Click on **Check Out** , then click **OK.**

d. Select the object in a directory viewer window and choose **Control:Change Object Properties.** The **Change Object Properties** dialog box is displayed. Click on the **Check Out** radio button. Click **OK.**

Only the highest numbered version of an object can be checked out. Checking the object out creates a new version with the next higher number and this is the version that can be changed.

To check in an object:

a. From a directory viewer window with the object selected, or from an editor window on the object itself, choose **Control:Check In.** This displays the **Check In** dialog box. Click OK.

b. From a directory viewer window with the object selected, or from an editor window on the object itself, choose **Control:Object Properties.** This displays an **Object Properties** dialog box. Click on **Check In,** then click **OK.**

c. From a directory viewer window with the object selected, or from an editor window on the object itself, choose **Control:Change Object Properties.** Click on **Check In,** then click **OK.**

6 Conclusions

The Ada programming language supports structured programming, top-down program development, re-use of general purpose components, and separate compilation of program components. Hence, it is the language that is the perfect choice for large software projects, while it is still efficient for average, or even, small projects. I am sure that it will be the only language used at establishments such as WES in the very near future.

Apex is designed to make software development in Ada an easy task. It has the facility to separate a large system into subsystems for team assignments in the development stage. The Ada language by nature allows these subsystems to be developed, compiled, debugged, and tested as separate units, then combined to perform the system task. Using **Apex,** the manager of a large project will have access to, and control over, each subsystem during development. Each developer will have access to his subsystem and any subsystem that is dependent upon his unit. **Apex** requires that a program unit be checked out by a developer and then checked in after editing it. Only one developer can edit a program unit at any given time. This way, the manager can monitor the progress of the project at any time during the development stage.

Apex has the ability to generate object code that will execute on a variety of hardware platforms.

References

Booch, Grady, Software Engineering With Ada. Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1983.

Caverly, Phillip and Goldstein, Phillip, Introduction to Ada: A Top-Down Approach for Programmers. Monterey, CA: Brooks/Cole Publishing Company, 1986.

Cohen, Norman H., Ada As A Second Language. New York: McGraw-Hill Book Company, 1986.

Dale, Nell, Weems, Chip and McCormick, John, Programming and Problem Solving With Ada. Lexington, MA: D. C. Heath and Company, 1994.

DeLillo, Nicholas J., A First Course in Computer Science With Ada. Homewood, IL: Richard D. Irwin, Inc., 1993.

Feldman, Michael B., Data Structures With Ada. Reading, MA: Addison-Wesley Publishing Company, 1993.

Gehani, Narain, UNIX Ada Programming. Englewood Cliffs: Prentice-Hall, Inc., 1987.

Rational Apex Compiler Reference Manual. Rational Software Corporation, Product Number 4000-00776, 1995.

Rudd, David, Introduction to Software Design and Development With Ada. St. Paul, MN: West publishing Company, 1995.

Texel, Putnam P., Introductory Ada: Packages for Programming. Belmont, CA: Wadsworth Publishing Company, 1986.

Tremblay, Jean-Paul, DeDourek, John M. and Friesen, Verna J., Programming in Ada. New York: McGraw-Hill Publishing Company, 1990.

Using Rational Apex. Rational Software Corporation, Product Number 4100-00285, 1995.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** ***(Leave blank)***	2. **REPORT DATE** February 1997	3. **REPORT TYPE AND DATES COVERED** Final report

4. TITLE AND SUBTITLE
A Brief Study of Rational Apex

5. FUNDING NUMBERS

6. AUTHOR(S)
Clyde Christopher

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Engineer Waterways Experiment Station
3909 Halls Ferry Road, Vicksburg, MS 39180-6199

8. PERFORMING ORGANIZATION REPORT NUMBER
Technical Report ITL-97-2

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Corps of Engineers
Washington, DC 20314-1000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Available from National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.
This report supersedes Technical Report ITL-96-4.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

The programming language Ada was developed by the U.S. Department of Defense to help control the increasing cost of its software. It is fast becoming the major general-purpose computer language. The language features type declarations, constrained and unconstrained array processing, and packages. It supports modern software engineering methods such as structured programming, data abstraction, top-down program development, re-use of general-purpose components, and separate compilation of program components.

Apex is an easy-to-use Ada software-engineering environment, developed by Rational Software Corporation, that helps to develop large, complex projects in a minimum time frame. Some features of **Apex** are as follows: (a) rational subsystems for architectural control; (b) rational CMVC, a sophisticated system for configuration and version control; (c) an intelligent Ada editor; (d) tools for Ada compilation management; (e) an Ada debugger; (f) a modern graphical user interface; (g) tools for customizing the **Apex** user interface; and (h) features for integrating **Apex** with other tools.

14. SUBJECT TERMS

Ada	Configuration management version control
Apex	Waterways Experiment Station

15. NUMBER OF PAGES
36

16. PRICE CODE

17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT**	20. **LIMITATION OF ABSTRACT**